

anki_vector.world

Vector's known view of his world.

This view includes objects and faces it knows about and can currently see with its camera.

Classes

<code>World</code> (robot)	Represents the state of the world, as known to Vector.
----------------------------	--

class `anki_vector.world.World(robot)`

Represents the state of the world, as known to Vector.

`all_objects`

yields each object in the world.

```
# Print the all objects' class details
import anki_vector
with anki_vector.Robot() as robot:
    for obj in robot.world.all_objects:
        print(obj)
```

Returns

A generator yielding `anki_vector.faces.Face`, `anki_vector.faces.LightCube`, `anki_vector.faces.Charger`, `anki_vector.faces.CustomObject` and `anki_vector.faces.FixedCustomObject` instances.

Type

generator

`charger`

Returns the most recently observed Vector charger object, or None if no chargers have been observed.

```
import anki_vector

# First, place Vector directly in front of his charger so he can observe it.

with anki_vector.Robot() as robot:
    print('Most recently observed charger: {}'.format(robot.world.charger))
```

Return type

Charger

charger_factory

alias of `anki_vector.objects.Charger`

close()

The world will tear down all its faces and objects.

connect_cube()

Attempt to connect to a cube.

If a cube is currently connected, this will do nothing.

```
import anki_vector

with anki_vector.Robot() as robot:
    robot.world.connect_cube()
```

Return type

ConnectCubeResponse

connected_light_cube

A light cube connected to Vector, if any.

```
import anki_vector

with anki_vector.Robot() as robot:
    robot.world.connect_cube()
    if robot.world.connected_light_cube:
        dock_response = robot.behavior.dock_with_cube(robot.world.connected_light_cube)
```

Return type

LightCube

`create_custom_fixed_object(pose, x_size_mm, y_size_mm, z_size_mm, relative_to_robot=False, use_robot_origin=True)`

Defines a cuboid of custom size and places it in the world. It cannot be observed.

See `objects.CustomObjectMarkers`.

Parameters

- **pose** (`Pose`) – The pose of the object we are creating.
- **x_size_mm** (`float`) – size of the object (in millimeters) in the x axis.
- **y_size_mm** (`float`) – size of the object (in millimeters) in the y axis.
- **z_size_mm** (`float`) – size of the object (in millimeters) in the z axis.
- **relative_to_robot** (`bool`) – whether or not the pose given assumes the robot's pose as its origin.
- **use_robot_origin** (`bool`) – whether or not to override the origin_id in the given pose to be the origin_id of Vector.

```
import anki_vector
from anki_vector.util import degrees, Pose

with anki_vector.Robot() as robot:
    robot.world.create_custom_fixed_object(Pose(100, 0, 0, angle_z=degrees(0)),
                                           x_size_mm=10, y_size_mm=100, z_size_mm=100,
                                           relative_to_robot=True)
```

Returns

FixedCustomObject instance with the specified dimensions and pose.

Return type

`FixedCustomObject`

custom_object_archetypes

yields each custom object archetype that Vector will look for.

See `objects.CustomObjectMarkers`.

```
import anki_vector
with anki_vector.Robot() as robot:
    for obj in robot.world.custom_object_archetypes:
        print(obj)
```

Returns

A generator yielding CustomObjectArchetype instances

Return type

`Iterable` [`CustomObjectArchetype`]

Type

custom_object_factory

alias of `anki_vector.objects.CustomObject`

define_custom_box(*custom_object_type, marker_front, marker_back, marker_top, marker_bottom, marker_left, marker_right, depth_mm, width_mm, height_mm, marker_width_mm, marker_height_mm, is_unique=True*)

Defines a cuboid of custom size and binds it to a specific custom object type.

The robot will now detect the markers associated with this object and send an `object_observed` message when they are seen. The markers must be placed in the center of their respective sides. All 6 markers must be unique.

See `objects.CustomObjectMarkers`.

Parameters

- **custom_object_type** (`CustomObjectTypes`) – the object type you are binding this custom object to.
- **marker_front** (`CustomObjectMarkers`) – the marker affixed to the front of the object.
- **marker_back** (`CustomObjectMarkers`) – the marker affixed to the back of the object.
- **marker_top** (`CustomObjectMarkers`) – the marker affixed to the top of the object.
- **marker_bottom** (`CustomObjectMarkers`) – the marker affixed to the bottom of the object.
- **marker_left** (`CustomObjectMarkers`) – the marker affixed to the left of the object.
- **marker_right** (`CustomObjectMarkers`) – the marker affixed to the right of the object.
- **depth_mm** (`float`) – depth of the object (in millimeters) (X axis).
- **width_mm** (`float`) – width of the object (in millimeters) (Y axis).
- **height_mm** (`float`) – height of the object (in millimeters) (Z axis). (the height of the object)
- **marker_width_mm** (`float`) – width of the printed marker (in millimeters).
- **marker_height_mm** – height of the printed marker (in millimeters).
- **is_unique** (`bool`) – If True, the robot will assume there is only 1 of this object. (and therefore only 1 of each of any of these markers) in the world.

```

import anki_vector
with anki_vector.Robot(enable_custom_object_detection=True) as robot:

    robot.world.define_custom_box(custom_object_type=anki_vector.objects.CustomObjectTypes.Cus

                                marker_front=
anki_vector.objects.CustomObjectMarkers.Circles2,
                                marker_back=
anki_vector.objects.CustomObjectMarkers.Circles3,
                                marker_top=
anki_vector.objects.CustomObjectMarkers.Circles4,
                                marker_bottom=
anki_vector.objects.CustomObjectMarkers.Circles5,
                                marker_left=
anki_vector.objects.CustomObjectMarkers.Triangles2,
                                marker_right=
anki_vector.objects.CustomObjectMarkers.Triangles3,
                                depth_mm=20.0, width_mm=20.0, height_mm=20.0,
                                marker_width_mm=50.0, marker_height_mm=50.0)

```

Returns

CustomObject instance with the specified dimensions. This is None if the definition failed internally. Note: No instances of this object are added to the world until they have been seen.

Raises

- **TypeError** if the `custom_object_type` is of the wrong type. –
- **ValueError** if the 6 markers aren't unique. –

Return type

CustomObject

define_custom_cube(*custom_object_type, marker, size_mm, marker_width_mm, marker_height_mm, is_unique=True*)

Defines a cube of custom size and binds it to a specific custom object type.

The robot will now detect the markers associated with this object and send an `object_observed` message when they are seen. The markers must be placed in the center of their respective sides.

See `objects.CustomObjectMarkers` .

Parameters

- **custom_object_type** (`CustomObjectTypes`) – the object type you are binding this custom object to.
- **marker** (`CustomObjectMarkers`) – the marker affixed to every side of the cube.

- `size_mm ( float )` – size of each side of the cube (in millimeters).
- `marker_width_mm ( float )` – width of the printed marker (in millimeters).
- `marker_height_mm` – height of the printed marker (in millimeters).
- `is_unique ( bool )` – If True, the robot will assume there is only 1 of this object (and therefore only 1 of each of any of these markers) in the world.

```
import anki_vector
with anki_vector.Robot(enable_custom_object_detection=True) as robot:

    robot.world.define_custom_cube(custom_object_type=anki_vector.objects.CustomObjectTypes.Cu

    marker=anki_vector.objects.CustomObjectMarkers.Circles2,
                                size_mm=20.0,
                                marker_width_mm=50.0, marker_height_mm=50.0)
```

Returns

CustomObject instance with the specified dimensions. This is None if the definition failed internally. Note: No instances of this object are added to the world until they have been seen.

Raises

TypeError if the `custom_object_type` is of the wrong type. –

Return type

`CustomObject`

define_custom_wall(*custom_object_type, marker, width_mm, height_mm, marker_width_mm, marker_height_mm, is_unique=True*)

Defines a wall of custom width and height, with a fixed depth of 10mm, and binds it to a specific custom object type.

The robot will now detect the markers associated with this object and send an `object_observed` message when they are seen. The markers must be placed in the center of their respective sides.

See `objects.CustomObjectMarkers` .

Parameters

- `custom_object_type ( CustomObjectTypes )` – the object type you are binding this custom object to.

- **marker** (`CustomObjectMarkers`) – the marker affixed to the front and back of the wall.
- **width_mm** (`float`) – width of the object (in millimeters). (Y axis).
- **height_mm** (`float`) – height of the object (in millimeters). (Z axis).
- **width_mm** – width of the wall (along Y axis) (in millimeters).
- **height_mm** – height of the wall (along Z axis) (in millimeters).
- **marker_width_mm** (`float`) – width of the printed marker (in millimeters).
- **marker_height_mm** – height of the printed marker (in millimeters).
- **is_unique** (`bool`) – If True, the robot will assume there is only 1 of this object (and therefore only 1 of each of any of these markers) in the world.

```
import anki_vector
with anki_vector.Robot(enable_custom_object_detection=True) as robot:

    robot.world.define_custom_wall(custom_object_type=anki_vector.objects.CustomObjectTypes.Cu

    marker=anki_vector.objects.CustomObjectMarkers.Circles2,
                                width_mm=20.0, height_mm=20.0,
                                marker_width_mm=50.0, marker_height_mm=50.0)
```

Returns

CustomObject instance with the specified dimensions. This is None if the definition failed internally. Note: No instances of this object are added to the world until they have been seen.

Raises

TypeError if the `custom_object_type` is of the wrong type. –

Return type

`CustomObject`

delete_custom_objects(*delete_custom_marker_objects=True, delete_fixed_custom_objects=True, delete_custom_object_archetypes=True*)

Causes the robot to forget about custom objects it currently knows about.

See `objects.CustomObjectMarkers` .

```
import anki_vector
with anki_vector.Robot() as robot:
    robot.world.delete_custom_objects()
```

disconnect_cube()

Requests a disconnection from the currently connected cube.

```
import anki_vector

with anki_vector.Robot() as robot:
    robot.world.disconnect_cube()
```

Return type

DisconnectCubeResponse

face_factory

alias of `anki_vector.faces.Face`

fixed_custom_object_factory

alias of `anki_vector.objects.FixedCustomObject`

flash_cube_lights()

Flash cube lights

Plays the default cube connection animation on the currently connected cube's lights.

```
import anki_vector

with anki_vector.Robot() as robot:
    robot.world.flash_cube_lights()
```

Return type

FlashCubeLightsResponse

forget_preferred_cube()

Forget preferred cube.

'Forget' the robot's preferred cube. This will cause the robot to connect to the cube with the highest RSSI (signal strength) next time a connection is requested.

```
import anki_vector

with anki_vector.Robot() as robot:
    robot.world.forget_preferred_cube()
```

Return type

ForgetPreferredCubeResponse

get_face(face_id)

Fetches a Face instance with the given id.

```
import time

import anki_vector
from anki_vector.events import Events
from anki_vector.util import degrees

def test_subscriber(robot, event_type, event):
    for face in robot.world.visible_faces:
        print(f"Face id: {face.face_id}")

with anki_vector.Robot(enable_face_detection=True) as robot:
    # If necessary, move Vector's Head and Lift to make it easy to see his face
    robot.behavior.set_head_angle(degrees(45.0))
    robot.behavior.set_lift_height(0.0)

    robot.events.subscribe(test_subscriber, Events.robot_changed_observed_face_id)
    robot.events.subscribe(test_subscriber, Events.robot_observed_face)

    print("----- show vector your face, press ctrl+c to exit early -----")
    try:
        time.sleep(5)
    except KeyboardInterrupt:
        robot.disconnect()
```

Return type

Face

get_object(object_id)

Fetches an object instance with the given id.

```
import anki_vector

with anki_vector.Robot() as robot:
    # First get an existing object id, for instance:
    valid_object_id = 1

    # Then use the object_id to retrieve the object instance:
    my_obj = robot.world.get_object(valid_object_id)
```

light_cube

Returns the vector light cube object, regardless of its connection status.

```
import anki_vector
with anki_vector.Robot() as robot:
    cube = robot.world.light_cube
    if cube:
        if cube.is_connected:
            print("LightCube is connected.")
        else:
            print("LightCube isn't connected")
```

Raises

ValueError –

Return type

LightCube

light_cube_factory

alias of `anki_vector.objects.LightCube`

set_preferred_cube(*factory_id*)

Set preferred cube.

Set the robot's preferred cube and save it to disk. The robot will always attempt to connect to this cube if it is available. This is only used in simulation (for now).

```
import anki_vector

with anki_vector.Robot() as robot:
    connected_cube = robot.world.connected_light_cube
    if connected_cube:
        robot.world.set_preferred_cube(connected_cube.factory_id)
```

Parameters

factory_id (`str`) – The unique hardware id of the physical cube.

Return type

SetPreferredCubeResponse

visible_custom_objects

yields each custom object that Vector can currently see.

See `objects.CustomObjectMarkers` .

```
import anki_vector
with anki_vector.Robot() as robot:
    for obj in robot.world.visible_custom_objects:
        print(obj)
```

Returns

A generator yielding CustomObject instances

Return type

Iterable [CustomObject]

Type

generator

visible_faces

yields each face that Vector can currently see.

```
import time

import anki_vector
from anki_vector.events import Events
from anki_vector.util import degrees

def test_subscriber(robot, event_type, event):
    print(f"Subscriber called for: {event_type} = {event}")

    for face in robot.world.visible_faces:
        print("--- Face attributes ---")
        print(f"Face id: {face.face_id}")
        print(f"Updated face id: {face.updated_face_id}")
        print(f"Name: {face.name}")
        print(f"Expression: {face.expression}")
        print(f"Timestamp: {face.last_observed_time}")
        print(f"Pose: {face.pose}")
        print(f"Image Rect: {face.last_observed_image_rect}")
        print(f"Expression score: {face.expression_score}")
        print(f"Left eye: {face.left_eye}")
        print(f"Right eye: {face.right_eye}")
        print(f"Nose: {face.nose}")
        print(f"Mouth: {face.mouth}")

    with anki_vector.Robot(enable_face_detection=True) as robot:
        # If necessary, move Vector's Head and Lift to make it easy to see his face
        robot.behavior.set_head_angle(degrees(45.0))
        robot.behavior.set_lift_height(0.0)

        robot.events.subscribe(test_subscriber, Events.robot_changed_observed_face_id)
        robot.events.subscribe(test_subscriber, Events.robot_observed_face)

    print("----- show vector your face, press ctrl+c to exit early -----")
    try:
        time.sleep(10)
    except KeyboardInterrupt:
        robot.disconnect()
```

Returns

A generator yielding `anki_vector.faces.Face` instances.

Return type

`Iterable[Face]`

Type

generator

visible_objects

yields each object that Vector can currently see.

```
import anki_vector
with anki_vector.Robot() as robot:
    for obj in robot.world.visible_objects:
        print(obj)
```

Returns

A generator yielding Charger, LightCube and CustomObject instances

Return type

`Iterable[ObservableObject]`

Type

generator